The FLiPSiDE Blackboard: A Financial Logic Programming System for Distributed Expertise

David G. Schwartz and Leon S. Sterling

Dept. of Computer Engineering & Science and Center for Automation & Intelligent Systems Research,
Case Western Reserve University Cleveland, OH 44106

Abstract

The blackboard model of problem solving is applied to the domain of portfolio management. A client-server blackboard architecture to control and integrate heterogeneous knowledge sources is presented. Software engineering benefits of the general approach, such as modularity, incremental development and ease of maintenance are discussed. Advantages of working with distributed, high granularity knowledge sources are illustrated. An explicit control mechanism for opportunistic knowledge source scheduling is presented as an integral part of the architecture.

1 Introduction

A portfolio manager is faced with a daunting information management task. We are addressing the task of building computer support systems for portfolio managers, and in particular the use of the blackboard control architecture for such a system. The blackboard control paradigm is an important contribution to computer science that has arisen out of the AI work of the previous decade. Our application of this paradigm is motivated by the complexity of portfolio management and the architectural strengths of the blackboard paradigm. In this paper we outline portfolio management and explain how a blackboard approach is suitable for this application. We then present our blackboard implementation with descriptions of the major domain-related and control-related components of the FLiPSiDE system. We include representative portions of Prolog code.

For our purposes a portfolio manager plans and executes trades which optimally implement a given business strategy with regard to the (utility based) trade-off to be maintained between risks and rewards. We group into four categories his activities in carrying out this task:

1. Monitoring the state of the market.
2. Monitoring the state of the current portfolio.
3. Maintaining his view of future market conditions.
4. Planning and executing the trades.

In terms of these categories we can observe some characteristics of the corresponding information management tasks.

Market information, for example, can be available in great volume and fine granularity, possibly requiring realtime buffering, filtering, or aggregation and representation in terms of parametric models. Data sources may be intermittent, physically dispersed, etc., requiring that the market itself be modelled by realtime interpolation or other estimation techniques. A generic problem is focus of attention for the manager.

Similarly, the portfolio may be traded by multiple agents, with different temporal reporting characteristics. The evaluation of risk/reward characteristics may be computationally very expensive, especially if positions are very dynamic or if they involve many derivative instruments. Portfolio monitoring includes consideration of current positions held, evaluating the aggregate response of the portfolio to changes in market conditions, and estimating the expected change in the portfolio over time.

The portfolio manager’s view will change over time and may be based on what she considers to be her “superior knowledge”, (or based on the realization that...
there is no superior knowledge). Planning and execution systems may have high or low level conceptual interfaces, provide for automatic execution of trades, or interact directly with back office systems.

When considering the task of building support systems for portfolio management, we identify two primary architectural concerns. We assume that the task will be divided across multiple subsystems supporting various subsets of these activities. We also observe that the tasks are interrelated (for example, the market focus of attention will be affected by the current positions held). Therefore the first issue is the task of integrating the information flow between these subsystems. This task is compounded by the wide variety of subtasks in terms of data representation, level of data abstraction, temporal response characteristics, etc. The second concern is for the task of controlling the subsystems.

We mention that the architecture must provide flexible mechanisms for performing this integration and control. At any point in time, in a given portfolio management environment, there are likely to be existing subsystems which are not candidates for rewriting or redesign, some subsystems under development and/or redesign, and a number of subsystems or activities not supported or not integrated.

Another set of architectural issues is motivated by software engineering concerns. For example, following the principles of data abstraction we would want to require that subsystems interact among themselves only through fixed, well specified mechanisms.

To summarize, we seek a framework that addresses a number of problems associated with realtime financial decision making systems:

- Allow the integration of diverse subsystems over time, including both discrete and continuous streams of information.
- Support different models of knowledge representation and inference.
- Focus attention for maximum decision aid in a constrained time.
- Facilitate the separability of the components.
- Improve the effectiveness of the portfolio manager.

2 A Blackboard Approach

The original blackboard paradigm, as proposed by Newell in 1962 [14] can best be described by way of analogy. Envision six independent domain experts meeting in a conference room with a blackboard. Five of these experts (E1 - E5) are specialists in different but related domains and the sixth expert (E6) has the ability to synthesize the most brilliant statements of the other five and form the final solution to a problem. E6 begins the session by taking the single piece of chalk available and writing a question on the blackboard. Each expert, E1 - E5 reads the question and, if he has something constructive to say, asks for a turn to go to the board to write down his thoughts. As each new thought is written on the blackboard, the individual experts reevaluate their opinions with respect to the new information, possibly requesting to modify something they have previously written on the blackboard. This continues until either (a) E6 synthesizes a solution to the original question - success or (b) none of the experts have anything more to write on the blackboard - failure.

The blackboard model of problem solving [14] has emerged from the original paradigm as a framework to coordinate disjoint systems. Blackboard architectures have been applied to a number of problems involving heterogeneous knowledge sources and realtime decision constraints, primarily in the areas of signal understanding, be it in regards to Speech (HEARSAY) [12], Sonar (HASP) [15], or Military Field Data (TRICERO) [18]. We believe that the success of blackboard architectures in signal understanding, and more recently in planning problems [8], can be applied to the portfolio management problem as described above.

The characteristics of blackboard systems that indicate its relevance to our domain are:

1. Blackboards can be used to integrate subsystems which use widely divergent representations of information. This information can be used at multiple levels of abstraction, and include both continuous and discrete streams of information. A blackboard architecture can integrate subsystems, or knowledge sources, that employ different models of computation and inference.

2. Incremental implementation and ease of maintenance are of vital importance. The abstraction of knowledge sources leads to a simplified engineering task and easier maintainability. The ability to modularize is a primary benefit of the architecture.

3. A blackboard system can control and integrate granulated, parallelized knowledge sources to
meet the time constraints imposed by realtime problems.

These three areas correspond to the problem areas of portfolio management identified in Section 1. Aside from the need to coordinate the activities of multiple heterogeneous knowledge-based components, portfolio management exhibits certain characteristics that are found to be common among many applications in the domain of financial analysis.

- The portfolio management task allows for flexible decomposition.
- Problem specifications of portfolio management subsystems tend to change.
- Realtime considerations indicate that an adequate temporal response is required.

For a more general enumeration of application characteristics common among blackboard systems the reader is referred to [6, 9, 13].

The first stage of our research focuses on the control aspect of blackboard architectures. Rather than aiming for a system that reaches a higher level decision by combining the expert decision power of smaller subsystems, we seek to provide an environment in which these subsystems are utilized in the most effective manner. We require a knowledge-based control mechanism that determines which knowledge source to call at what time, and at what point to cut off the processing of one knowledge source in favor of another. This can be done by including heuristic knowledge source triggers and preconditions, and by having a dynamic reevaluation of the efficacy of each knowledge source. Thus we can take advantage of the flexibility of a control blackboard architecture to focus our processing, without requiring that the diverse subsystems actually achieve a higher goal or solve a common problem. By having a control system that is aware of the nature of the common problem, and can evaluate the ability of each knowledge source, we can provide the sophisticated user with automatic dynamic control of complex autonomous systems.

3 Prolog as a Platform

Large and complex systems tend to evolve over protracted periods of time. Blackboard systems in particular can be large and complex resulting in their being expensive to build and run [13]. In dealing with systems in which module specifications often change, the rapid prototyping made possible by the use of Prolog becomes very advantageous. The rapid prototyping is facilitated by using Prolog's predicate logic format to create clear representations, allowing translation and manipulation of symbolic structures [16].

Tight declarative coding can save an order of magnitude of code. Researchers in Prolog applications development report substantial coding reductions when compared to other languages. Reintjes [17] found a ten-fold reduction in the code required to implement a full-featured VLSI editing system. Drongowski's work in Prolog-based design systems [5] discusses a 300-line recursive-descent parser written in C which took only 75 lines to implement in Prolog. Despite these improvements in coding size, there was no deterioration of code readability. On the contrary, the decrease in code coupled with the declarative nature of Prolog leads to easier understanding of written code which translates directly into savings when it comes to maintenance of the code over time.

Prolog's builtin pattern matching capabilities greatly simplify the code necessary to implement knowledge source triggers and knowledge source preconditions. Using non-ground terms in trigger patterns provides generalized event matching behavior. In addition, builtin backtracking provides a ready made backward chaining inference engine. These characteristics are discussed by Jones et al in their report on a blackboard shell in Prolog [11]. Our approach differs from theirs significantly in its support of heterogeneous subsystems, consideration of software engineering and control issues, and use of a distributed client-server architecture. Bisiani and Forin [2] suggest that a parallel Prolog shows potential for blackboard applications because of its natural tendency to promote OR-parallelism, and its builtin database facility that can easily support blackboard data structures. The considerable synergy to be found between blackboard architectures and parallel Prolog environments such as Shared Prolog is discussed in [3].

Our choice of BIM ProLog as a development language was based on its support of multiple external database systems, and other logistic considerations specific to our application. In the text that follows we use a different font to denote Prolog code. For clarity we have chosen to represent knowledge items as Prolog atoms with names corresponding to their natural English usage, for example the concept "financial feasibility" appears as the atom financial_feasibility.
4 The FLiPSiDE Architecture

4.1 Applying the Blackboard Model

Before describing the chosen blackboard architecture, let us introduce the main components of any simple blackboard architecture. A basic blackboard system is comprised of three parts: Knowledge Sources, a Blackboard Data Structure, and a Control Mechanism. These components can be implemented on a serial machine with the controller scheduling each knowledge source for its turn using the processor. Each knowledge source combines some form of inference and knowledge base to solve a part of the overall problem. The Blackboard Data Structure serves as a repository for data items produced or deduced by any knowledge source whose value may be of interest to other knowledge sources. The Control Mechanism has the task of deciding which knowledge source should be invoked at what time.

A control blackboard architecture can maximize the effectiveness of the portfolio manager by coordinating the interaction between the portfolio manager and the multiple sources of information. The FLiPSiDE blackboard architecture takes some of its primary characteristics from the BB1 Blackboard Architecture for Control [8, 10]. The BB1 architecture distinguishes itself in its treatment of the control aspect of blackboard operation as a decision problem in its own right. Considering the problems that we wish to concentrate on, the BB1 architecture opens up a number of options. We require three main behavioral characteristics of our architecture:

- Treat control explicitly as opposed to trivially or implicitly.
- Integrate multiple control heuristics to cover varying aspects of the control problem.
- Reevaluate and modify the current control strategy in light of recent changes in the problem domain.

For other characteristics of control blackboard architectures the reader is referred to [8] for a detailed discussion.

4.2 Blackboard Servers and Knowledge Servers

Treating the blackboard data repository as a server for a number of knowledge sources was first implemented as part of GBB as discussed by Corkill in [4]. In our application two significant characteristics indicate the suitability of a blackboard server approach.

1. The network of knowledge sources is loosely coupled.
2. A shared memory space is not available.

Additional aspects of a server approach, such as maintaining only one copy of each blackboard item, are detailed in [4]. We have extended the client-server approach to encompass the actual knowledge sources. As in Corkill's model, each knowledge source is a client to one or more blackboard servers. But in addition, we can combine the design strengths of a control blackboard with the client-server model and treat each knowledge source as a server. Each serving knowledge source, or Knowledge Server, serves (possibly partial) solutions to a central blackboard coordinator process. The Blackboard Coordinator is also a client to a series of Control Knowledge Servers which determine the next goal to be sent to an appropriate Domain Knowledge Server (figure 1).

The client-server communication is handled by a standard IPC communication protocol. A knowledge source communicates with its server blackboard via the Prolog predicate bbservice/5. The knowledge source instantiates arguments identifying: the blackboard to be accessed; the type of operation to perform; the level of the blackboard to be searched; and the name of the item sought. In performing read-related operations the fifth argument is uninstantiated, whereas in a write-related operation the fifth argument contains the value of the item to be written.

A blackboard server is able to service six different requests:

1. read - Retrieve the value of an item from the blackboard.
2. write - Post a new item to the blackboard or modify the value of an existing item.
3. delete - Remove an item from the blackboard. This differs from performing a "write" with a null value in that the item ceases to exist on the blackboard and subsequent existence tests will fail.
4. on - A computationally cheap test for the existence of an item on the blackboard without reading the item's value.
5. multi_read - Retrieve a number of items in a single server call.
6. multi_write - Write a number of items in a single server call.
4.3 Knowledge Sources

With the goal of supporting heterogeneous knowledge sources comes the burden of multiple language support. At present the native language of each knowledge source must be Prolog. By native language we mean the language level at which the knowledge source communicates with the main blackboard. This does not mean that the complete knowledge source is restricted to Prolog, but rather it is restricted to Prolog and those languages directly callable from the Prolog implementation being used. In our case this includes C, Pascal, Fortran, and Assembler all of which are supported by the BIM ProLog system [1]. Each knowledge source accesses the blackboard through a common Prolog-based language interface with predicates for writing to the blackboard and reading from the blackboard. The direct interface between a foreign language knowledge source and the Prolog blackboard would take the form of a front-end processor similar to what is described by Yang et al in [21]. The Knowledge Server to Blackboard Server interface forms the nucleus of an embedded language that is accessible to the knowledge sources.

Using Prolog’s meta-interpretation capabilities to develop application-specific embedded languages for expert systems is a powerful development technique. We can extend any embedded language with the appropriate blackboard access predicates to combine the strengths of meta-interpreters and blackboard architectures. Embedding this special-purpose functionality, in our case blackboard access, into Prolog meta-interpreters is dealt with in detail in [19] and [20].

Aside from interacting with the blackboard, each knowledge source has the ability to produce its own output directly to the user. This is an essential characteristic considering our goal of optimizing interaction between the user and the system. This is done in a manner similar to blackboard access using a Unified User Interface. The Unified User Interface is a server process that handles requests from Knowledge Servers to display messages and receive input from the user. A knowledge source interacts with the user through the uuiserve/4 predicate by instantiating arguments identifying: the user interface to be accessed; whether to perform an ask or tell operation; the message or prompt to be displayed; and in the case of ask an uninstantiated term to unify with the user’s response.

Knowledge sources employing different inference mechanisms are being used to test the flexibility of the KS-to-Blackboard interface. In our current application we have identified five domain knowledge sources. The first two of these are implemented as backward chaining meta-interpreters, and the third runs as an unstructured pure Prolog program. These three knowledge sources existed previously as standalone expert system programs and have been modified and integrated into FLiPSiDE. The last two knowledge sources are currently under development and include a forward-chaining inference model and a
C-language knowledge source to test foreign language performance.

4.4 FLiPSiDE's Blackboard Coordinator

The main loop of the blackboard coordinator will place the main goal onto the blackboard which in turn will generate an event. From that point on, a solution loop deals with progressive iterations of blackboard activity. In every iteration the system must perform certain fixed tasks. These include:

- Choose the first knowledge source on the agenda that passes its precondition. This is done by successively testing each knowledge source in the ranked agenda until a precondition is passed.
- Invoke the chosen knowledge source which will, in most instances, update the blackboard with new or changed items causing new events.
- Check the knowledge source triggers to see which other knowledge sources may be of use given the recent changes (events) to the blackboard.
- Update the agenda with the names of newly pending knowledge sources.
- Have the control mechanism rank the pending knowledge sources in the new agenda to determine the most suitable one to invoke.
- Recursively repeat the loop until the agenda is empty.

These basic steps are illustrated below in a portion of the Prolog code. Here pick..ks separates the initial agenda into the invocable knowledge source and the remainder of the initial agenda. Then invoke..ks executes the chosen knowledge source. check_triggers adds more knowledge source names to the remaining agenda to produce the next agenda to be dealt with. At this point the evaluate control subsystem is accessed to validate or update the current ranking criteria and rank the new agenda. Finally the same loop proceeds to process the ranked agenda.

```
bb_loop(Agenda) :- empty(Agenda).
bb_loop(Agenda) :-
    non_empty(Agenda),
    pick..ks(Agenda, Ks_to_do, RemainingAgenda),
    invoke..ks(Ks_to_do),
    check_triggers(RemainingAgenda, NextAgenda),
    evaluate.control(NextAgenda, RankedAgenda),
    bb_loop(RankedAgenda).
```

4.5 Knowledge Source Triggers

For a knowledge source to be eligible for invocation, a blackboard trigger indicating the need for the knowledge source must be activated. Knowledge source triggers are represented as simple Prolog facts of the form:

```
ks.trigger(KSNAME..l, Triggering.Event..1).
ks.trigger(KSNAME..l, Triggering.Event..2).
'...
ks.trigger(KSNAME..n, Triggering.Event..m).
```

This way the addition, deletion, and modification of knowledge source triggers is a simple matter of changing the appropriate knowledge source trigger facts. Each trigger condition should be a test of the occurrence of an event that has appeared on the blackboard in recent processing. Triggers are intended to provide a non-computationally-intensive way to decide on the basic eligibility of knowledge sources. An event is generated whenever any item is written to, deleted from, or modified on the blackboard. Events take the form of 3-element association lists such as: [[level, LevelID], [item, ItemID], [type, Type]]. Triggers can contain partially instantiated events for more general triggering. The trigger pattern [[level, indices], [item, dow30], [type, new]] will fire whenever the dow30 item is either newly written, modified, or deleted. The trigger pattern [[level, indices], [item, w], [type, new]] will fire when any new item is posted to the indices level of the blackboard.

Whenever ks.trigger(portfolio_evaluator, [[level,indices],[item,dow30],[type,new]]) is true, a request for invocation of the portfolio evaluation knowledge source will be placed on the agenda. Although such a request may appear quite often, it will not necessarily result in an invocation of portfolio_evaluator as the knowledge source must still pass its precondition.

4.6 Preconditions

Once a trigger has passed and a knowledge source has been chosen by the control mechanism as the best one to execute, a more complex precondition is tested. Only one precondition predicate is allowed for each knowledge source, but this precondition can be as simple or as complex as desired. A generic precondition is of the form: precondition(KSNAME) :- condition..1, condition..2,... condition..n. An example of a precondition using a nested if-then-else construct is:
precondition(decision) :-
  (decision_test(1) ; decision_test(2)).

decision_test(1) :-
  bbserve(dbb,read,decision,marketability,A),
  bbserve(dbb,read,sponsor,sponsor_quality,B),
  bbserve(dbb,read,income,income_quality,C).

decision_test(2) :-
  bbserve(dbb,read,decision,feasibility,A).

Here the “decision” knowledge source will be invoked if either: (a) the marketability, sponsor quality, and income quality items are available on their respective levels of the domain blackboard; or (b) the financial feasibility item is found on the decision level of the domain blackboard.

4.7 Knowledge Representation

One advantage to the FLiPSiDE blackboard server architecture is having the internal blackboard knowledge representation hidden from and transparent to the knowledge sources. When a knowledge source requests an item to be written to the blackboard using the bbserve(BlackboardID, write, Level, Item, Value) predicate, it does not need to be concerned with how the levels, items and values are actually stored. This allows us to experiment with different representations such as association lists, trees, and Prolog assertions without necessitating changes to the knowledge sources.

Our current implementation uses recorded association lists for each level of the blackboard. A level is recorded using its name as key and each level has a [(Item1, Value1), ... , (ItemN, ValueN)] structure. The term ValueN can be any arbitrary Prolog term, the only caveat being that all knowledge sources using that Item must be aware of the structure of the Value term.

Each knowledge source is made known to the system by a Prolog fact. For example the fact knowledge_server(portfolio_evaluator,bach,3000) names the portfolio evaluator as a knowledge source serving from port 3000 on machine bach. Similarly, blackboard servers are identified by blackboard_server(BlackboardAlias, Machine, Port) facts.

In order to coordinate the reference to identical items across different knowledge sources we use postable/2 facts. A postable(Item, Level) fact tells a knowledge source whether an item can be placed or found on the blackboard, and on what blackboard level. Postable/2 facts are instantiated locally in each Knowledge Server process upon startup and any changes or additions to these facts can be broadcast by the Blackboard Coordinator to all affected Knowledge Servers. This allows us to dynamically add new knowledge sources while the system is running, while maintaining consistency in blackboard references. It also makes it possible to improve blackboard access time by changing the blackboard level to which an item is assigned without having to change any code in the knowledge sources.

5 Control

5.1 The Control Problem

To opportunistically select the most favorable knowledge source for execution, a knowledge-based control mechanism is used. The structure of the control subsystem mirrors that of the domain problem solving subsystem. With its own blackboard for control knowledge items and multiple control knowledge servers, the control subsystem provides the same flexibility and engineering benefits that are enjoyed in the domain subsystem.

Why devote so many resources and so much effort to control? Control of multiple knowledge sources is a complex scheduling problem, and complex scheduling can be handled by a knowledge-based system no different than any other domain specific problem. Our control challenge goes beyond finding the knowledge source that can contribute the most in the next invocation. Aspects of our application domain require the ability to radically alter the method of knowledge source sequencing being used in order to respond to unexpected events in the outside world that feed into certain knowledge sources. We can use a generalized scheduler development environment as described in [7] in order to integrate diverse scheduling methodologies into scheduling programs. Having such a scheduler development environment allows timely experimentation with multiple scheduling methodologies.

5.2 The Control Blackboard

The Control Blackboard serves as a repository for control knowledge items. These items include:

- Empirical performance knowledge about the various physical machines on which the domain knowledge servers are running.
- Historical information tracking knowledge source invocations.
• Constraints affecting knowledge sources, such as estimated minimum execution times.

• Dynamically updated evaluations of knowledge source reliability and relevance depending on current progress towards a solution.

• Overriding policies for knowledge source selection such as: Run most critical; Run most time consuming; Run least time consuming and other scheduling heuristics.

The control evaluation loop is very similar to the domain problem solving loop. The one fundamental difference is that control knowledge sources are invoked in a predetermined sequence and not opportunistically. Were this not the case we would end up requiring a second control mechanism to handle the first controller, and so on ad infinitum.

5.3 The Control Knowledge Sources

We use four basic control knowledge sources, each of which is server based. The Evaluate Control knowledge source adjusts the relevance, efficiency, reliability and other information about the domain knowledge sources. It also reevaluates and adjusts the current overriding policies governing domain knowledge source ranking. The Trigger knowledge source is, as its name implies, a knowledge base of domain triggers which are checked against events. The Rank KS knowledge source applies the control knowledge to the list of triggered knowledge sources to produce a ranked list of domain knowledge sources. The Precondition knowledge source contains the more complex rules to be tested prior to knowledge source invocation, and applies these rules to the ranked knowledge source requests until the first eligible one is found.

6 Conclusions

We have identified Portfolio Management as a large and complicated decision problem, too complex to be effectively handled by a simple serial expert system. By taking advantage of the blackboard system paradigm we are realizing benefits of modular design and implementation. Our Knowledge Server approach combined with Blackboard Servers provides a distributed processing framework suitable for the deployment of large scale systems with heterogeneous knowledge sources. Future directions of this research point towards the reimplementation of the Blackboard Servers and Blackboard Coordinator to take advantage of the parallelism available in other logic programming languages. This will require a more complex knowledge representation scheme incorporating temporal information about knowledge source invocations and the resultant inferences. Even without parallelizing the knowledge sources there are potential gains to be had by simply parallelizing the Coordinator and blackboard access so that many serial knowledge sources can execute at the same time.

With each knowledge source implemented as a server we have the possibility of being able to distribute the blackboard across the server nodes providing each knowledge source with a local blackboard, as suggested in [4]. This can reduce traffic between knowledge sources and the central blackboard server. However, other knowledge sources must then know on which distributed blackboard an item may be found. This will add to the knowledge engineering and maintenance headache and defeat part of the purpose of using a blackboard approach in the first place. While distributing knowledge sources provides clear engineering advantages, the benefits of distributing the actual blackboard are not quite so obvious and demands further investigation.

For a system to take the FLiPSiDE approach there are two major application guidelines. First, the domain problem must be sufficiently large and complex to have the benefits of distributing and possibly parallelizing expertise outweigh the communications overhead imposed by the architecture. Second, the domain problem must lend itself to decomposition into large grained knowledge sources, or in the case of existing heterogeneous systems lend itself to a higher level integration of the problem solving process.

References


[4] D. D. Corkill. Design Alternatives for Parallel and Distributed Blackboard Systems. In V. Ja-


